University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

String Pattern Matching Algorithms

Raphael Ribeiro da Costa e Silva

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Carlos E. Ferreira

São Paulo

January 25, 2023

"The beauty of algorithms is that they are self-contained, timeless, and universal." - Edsger W. Dijkstra

Acknowledgements

I would like to express my deepest gratitude to those who have supported and encouraged me throughout my academic journey. I would like to thank my advisor Dr. Carlos E. Ferreira, for his guidance, knowledge, and support. I appreciate the time and effort he has invested in helping me develop and complete this monograph.

I would also like to thank my family, friends, and classmates who have offered me their love, support, and encouragement. Their belief in me has been a constant source of motivation throughout my studies.

I am grateful to the faculty and staff of the Institute of Mathematics and Statistics for providing me with a world-class education and the tools and resources necessary to succeed.

Thank you all for your support and for helping me to achieve this important milestone in my life.

Resumo

Raphael Ribeiro da Costa e Silva. **Algoritmos de Busca de Padrões em Strings**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O Pareamento Simples e Múltiplo de Strings é um dos problemas básicos de algoritmos em strings e vários algoritmos foram propostos para resolvê-lo. Existem soluções práticas para problemas reais que podem ser desenvolvidas usando esses algoritmos, incluindo, detecção de intrusos em sistemas, biologia evolucionária, linguística computacional e recuperação de dados. Nesse trabalho, estudamos diversos algoritmos propostos para resolver o problema de pareamento de strings, começando com pareamento simples e prosseguindo para discutir suas extensões para lidar com múltiplos padrões. Também foram conduzido experimentos para comparar a performance desses algoritmos.

Palavras-chave: Busca de Padrões. Strings. Pareamento.

Abstract

Raphael Ribeiro da Costa e Silva. **String Pattern Matching Algorithms**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

The String Pattern Matching Problem is one of the basic string algorithms problems and several algorithms have been proposed to solve it. There are practical solutions to real-world problems that can be developed using these algorithms, including, but not limited to, intrusion detection systems, evolutionary biology, computational linguistics, and data retrieval. In this paper, we shall study several algorithms proposed to solve the string matching problem, starting with single patterns and then proceeding to discuss its extensions to deal with multiple patterns. We also conduct experiments to compare the performance of these algorithms.

Keywords: Pattern Searching. Strings. Matching.

List of Abbreviations

BM	Boyer-Moore algorithm
BMH	Boyer-Moore-Horspool algorithm
TDS	Trie Data Structure
AHS	Aho-Corasick Algorithm
WMN	Wu-Manber algorithm
SPSM	Single Pattern String Matching Problem
MPSM	Multiple Pattern String Matching Problem

List of Symbols

$\mathcal{T}$	Text string
P	Pattern for the single pattern matching problem
$\mathcal{P}$	set of patterns for the multiple pattern matching problem
$\mathcal{A}$	an Alphabet of symbols

List of Figures

List of Tables

List of Programs

Contents

Chapter 1

Introduction

An **alphabet** Σ is a finite set. A **symbol** is an element $s \in \Sigma$. A **string** is a finite sequence of symbols, i.e, elements of an alphabet. The set of all strings over Σ is denoted by $\Sigma*$. We say a string $p \in \Sigma*$ is a **pattern** over a fixed alphabet Σ if p consists of symbols from Σ. Let s be a string and denote $s := s_1 \cdots s_n$. We say that a subsequence $s_i \cdots s_j$ of s is a **substring** of s. An **ocurrence** of u in s is a pair (i, j) such that $u := s_i \cdots s_j$ is a substring of s. A **prefix** of s is a substring $u := s_0 \cdots s_k$. i.e. s starts with u. A **suffix** of s is a substring $u := s_{n-k} \cdots s_{n-1}$. i.e. s ends with u. A **proper prefix** of s is a prefix u of s such that $u \neq s$. Moreover, a **proper suffix** of s is a suffix u of s such that $u \neq s$. A **border** of s is a substring u of s such that u is a proper suffix of s and u is a proper prefix of s.

Let P be a set of patterns over a fixed alphabet Σ and let T be a fixed string called text input. The Multiple Pattern String Matching (MPSM) is the problem of finding all occurrences of all patterns of P in T . The Single Pattern String Matching (SPSM) is a special case of (MPSM) by adding the constraint $|P| = 1$.

The MPSM is one of the basic string algorithms problems and several algorithms have been proposed to solve it. There are practical solutions to real-world problems that can be developed using these algorithms, including, but not limited to, intrusion detection systems, evolutionary biology, computational linguistics, and data retrieval. In this paper, we shall study the first algorithms proposed to the string matching problem, starting with single patterns and then proceeding to discuss its extensions to deal with multiple patterns. We shall study other algorithms and data structures for the multiple pattern string matching problem and then compare those algorithms regarding to the structure of the problem.

Chapter 2

Single Pattern String Matching

2.1 Introduction

In this chapter, we shall study algorithms that solve the following problem: Given a text $\mathcal{T}$ and a pattern P, both with symbols from the alphabet $\mathcal{A}$, find all occurences of P in $\mathcal{T}$.
We discuss the algorithms of Boyer-Moore and its heuristics; the Boyer-Moore-Horspool algorithm, which is a simplification of the original Boyer-Moore; and the Trie Data Structure. After this chapter, when we proceed to the multiple pattern problem, we will see further generalizations of those same algorithms in order to deal with multiple patterns.

2.2 Brute Force

Let T be a input text and P a pattern. A naive algorithm for finding all occurrences of P in T works as follows: we compare the text with pattern from left to right. At iteration i, we compare $P[0]$ with $T[i]$. If the first character is matched, then we need to check whether the remaining pattern is matched or not. Check the table 2.1

Program 2.1 Brute force search

```
1   FUNCTION BRUTEFORCEsearch(P, T)   ▷ returns the set of ocurrences of P in T
2     for i := 0, i < |T|, i ⟵ i + 1
3         match ⟵ true
4         for j := 0, j < |P|, j ⟵ j + 1
5             if T[i + j] ≠ P[j]
6                 match ⟵ false
7                 break
8         if match
9             ans ⟵ ans ∪ {i}
10        return ans
```

In worst case, the comparison in line 4 holds true for every iteration and the loop in line 5 is executed. So, the time complexity is $\mathcal{O}(mn)$. The space complexity is $\mathcal{O}(1)$ since we just need some flags beside the sizes of P and T themselves.

2.3 Boyer-Moore

We can improve the performance of the brute force algorithm by using some heuristics but yet using the same idea. Instead of comparing character by character, the Boyer-Moore Algorithm Boyer and Moore, 1977 introduces the bad character heuristic and the good suffix heuristic in order to shift the pattern.

2.3.1 Bad Character

When a mismatch occurs, we have to shift the pattern until we no longer have a mismatch, i.e, until we found a match, or until the pattern move past the mismatched character. The Bad Character rule tells us to shift the pattern until we align it with the rightmost occurrence of the mismatched character. We can see some examples:

Example

	0	1	2	3	4	5	6	7	8	9
T	a	b	b	a	d	a	b	a	c	b
P	a	b	c	a	b	**c**				
shift			a	b	c	a	b	c		

Table 2.1: *In this example, a mismatch occurs when comparing a-c. Then, we shift the pattern so that the letter a it is aligned with the rightmost a in P.*

Example

	0	1	2	3	4	5	6	7	8	9
T	a	b	b	**d**	a	d	b	a	c	b
P	a	b	c	**a**						
shift					a	b	c	a		

Table 2.2: *In this example, a mismatch occurs when comparing d-a. But there is no occurrence of the mismatched character in p. So we shift the pattern past the mismatched character*

To summarize, the Bad Character rule tells us to shift the pattern until the text is aligned with the righmost occurence of the mismatched character. We can simply calculate the righmost occurence x for each character, and, during the search phase, we have to shift the pattern by $x - s$ where s is the number of matched characters.

We are going to build a table BC for the bad character heuristic. To do so, we need to map for each symbol $s \in \mathcal{A}$, the index of its rightmost occurrence in P, or -1 if s does not occur in P. Therefore, the program 2.2 code builds the BC table

Program 2.2 Boyer-Moore BC Table.

```
FUNCTION buildBC(T,P)   ▷ returns the BC Table
  for s ∈ 𝒜
      bc[s] = −1
  end
  j ⟵ 0
  for c ∈ P
      bc[c] = j
      j ⟵ j +1
  end
  return bc
```

2.3.2 Good Suffix

The second rule we are goint to apply is the Good Suffix rule and it is applied regarding the borders of the pattern. A **border** is a substring of p that is both a proper suffix and a proper prefix of p. Let t be the substring of $\mathcal{T}$ which is matched to the pattern at some iteration i and we have found a mismatch. We can safely shift the pattern until t is aligned with the rightmost ocurrence of $t \in \mathcal{P}$. See this example:

Example

	0	1	2	3	4	5	6	7	8	9	10	11
T	a	b	c	a	b	c	**a**	**d**	**a**	a	b	c
P	b	a	d	a	a	a	**a**	**d**	**a**			
shift						b	a	d	a	a	a	a

Table 2.3: *A mismatch occurs comparing c-a. In this case, t = ada. We can shift the pattern until it is aligned with the rightmost occurence of ada, which occurs at index 1.*

Notice that this is only possible if $\mathcal{P}$ contains at least one other occurrence of t. When it is not the case, we can try a different idea: we can try to match a suffix of t with some prefix of $\mathcal{P}$. Check this example:

Example

	0	1	2	3	4	5	6	7	8	9	10	11
T	a	b	c	a	b	c	**a**	**d**	**a**	a	b	c
P				d	a	a	**a**	**d**	**a**			
shift								d	a	a	a	d

Table 2.4: *In this case, t = ada. There is no other occurence of ada in $\mathcal{P}$, but we can align the suffix da of t with the prefix ba of P*

So how do we apply the Good Suffix Rule?

We are going to build two tables: f and s. $f[i]$ will store the starting index of the widest border of $p[i \cdots |p| - 1]$. The table $s[i]$, on the other hand, will store the shift position for $s[i \cdots$. So, how do we build these tables? Let us build by induction, in decreasing order, for $i = m$, we define $f[i] = m + 1$. Now, suppose for some index $i < m$ we have $f[i] = j$, let us calculate $f[i - 1]$. Denote widest border for $p[i \cdots |p| - 1]$ as x and let us construct the widest border for $p[i - 1 \cdots |p| - 1$, namely y. If $s[i - 1] = s[j - 1]$ then we can define $y := s[i - 1] + x$. Therefore, $f[i - 1] = j - 1$.
If $s[i - 1]! = s[j - 1]$, then we set $j = f[j]$ and try again, while $j > m$. While doing our search, whenever we find $s[i] == s[j]$ and $s[i - 1]! = s[j - 1]$ we can determine shift for $s[j \cdots |p| - 1]$ matching and mismatch at at index $j - 1$, so $s[j] = j - i$.

2.3.3 Preprocessing

Example

Let $P = baababa$. Let's construct the tables f and s.

The suffix beginning at position 0 is baababa and it has border ba, starting at index 5. So, $f[0] = 5$
The suffix beginning at position 1 is aababa ant it has border a, starting at index 6. So $f[1] = 6$
The suffix beginning at position 2 is ababa has borders aba, starting at index 4 and b, starting at index 6. Since aba is the widest, $f[2] = 4$. However, notice that the border aba cannot be extended to the left, because $P[1] = a \neq b = P[3]$. So, $s[4] = 4 - 2 = 2$. Also, the border a cannot be extended either. So, $s[6] = 6 - 2 = 4$.
The suffix beginning at position 3 is baba and it has border ba, starting at index 5. Thus, $f[3] = 5$
The suffix beginning at position 4 is aba ant it has border a, starting at index 6. So $f[4] = 6$
The suffix beginning at position 5 is ba and it has no border. So, $f[5] = 7$
The suffix beginning at position 6 is a and it has empty border. Moreover, this border cannot be extended to the left. Therefore, $f[6] = 7$ and $s[6] = 7 - 6 = 1$. Notice that assigning this way ensure a valid shift.

	0	1	2	3	4	5	6
P	b	a	a	b	a	b	a
f	5	6	4	5	6	7	7
s	0	0	0	2	0	4	1

Table 2.5: *Tables f and s of Boyer-Moore Algorithm for $P = baababa$.*

In the second phase of the Good Suffix heuristic, a part of the matching suffix occurs at the beginning of P. Thus, we have a border of the pattern. So, we can shift it accordingly to its widest border. We shall determine the widest border of P that is contained in every suffix. At the end of phase 2, all values of s are determined.

We start with the widest border of the pattern, which is stored in $f[0]$ and change it if the current suffix becomes shorter than $f[0]$. The Program 2.3 shows the initialization of the Boyer-Moore algorithm.

Program 2.3 Boyer-Moore initialization.

```
FUNCTION BMinit(P,T)   ▷ return tables f and s
  i := |P|
  j := i + 1
  f[i] := j
  ▷ First phase
  while i > 0
      while j ≤ |P| and P[i − 1] ≠ P[j − 1]
          if s[j] == 0
              s[j] := j − i
          end
          j := f[j]
      end
      i := i − 1
      j := j − 1
      f[i] := j
  end
  ▷ Second phase
  j := f[0]
  for i := 0,i ≤ |P|, i := i + 1
      if s[i] == 0
          s[i] = j
      end
      if i == j
          j = f[j]
      end
  end
  return f, s
```

2.3.4 Search

For searching patterns, we compare symbols of pattern from right to left with text. If a match happens, the pattern is shifted accordingly to its widest border. Otherwise, the shift is determined by the maximum of the values given by the good-suffix and the bad-character heuristics. The program 2.4 shows the search phase of the Boyer-Moore Algorithm.

Program 2.4 Boyer-Moore search.

```
FUNCTION BMsearch(P,T,s,bc)   ▷ return occurrences of P in T, given tables s and bc.
  ans ⟵ ∅
  i := 0
  while i ≤ |T| − |P|
```

cont ⟶

```
⟶ cont
        j ⟵ |P| − 1
        while j ≥ 0 and P[j] == T[i + j]
            j := j − 1
        end
        if j < 0
            ans ⟵ ans ∪ {(i, i + |P| − 1)}
            i := i + s[0]
        end
        else
            i ⟵ max(s[j+1], j−bc[t[i+j]])
    end
    return ans
```

2.3.5 Complexity

The best case occurs when at each attempt the text character compared does not occur in the pattern and the pattern is shifted. In such case the algorithm runs in $\mathcal{O}(|\mathcal{T}|/|\mathcal{P}|)$. However, in general $O(|P||T|)$ comparisons are needed.

2.4 Boyer-Moore-Horspool Algorithm.

Instead of using the bad character and the good suffix heuristics. Horspool Horspool, 1980 proposed a simplified version of the Boyer-Moore algorithm. The idea is comparing the last character of the pattern $\mathcal{P}$ with the last character of text $\mathcal{T}$. If a mismatch occurs, then we shift the pattern to align it to the rightmost occurrence of the mismatched character. Otherwise we simply continue comparing the characters of $\mathcal{P}$ from right to left.

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12
$\mathcal{T}$	d	b	d	a	c	a	b	<u>c</u>	a	d	a	d	b
$\mathcal{P}$					b	c	a	<u>b</u>					
shift							b	c	a	b			

Table 2.6: *In this case, a mismatch occurs comparing c-b. So we shift the pattern to align it with the next rightmost occurrence of c in P*

Example

2.4.1 Preprocessing

The preprocessing phase of the algorithm is based on maping for each symbol $s \in \mathcal{A}$, the rightmost occurrence of s in $p[0 : m-2]$ i.e, the rightmost occurrence of s in the pattern except the last character. We initialize the table with $|p|$. Notice that if there is a

	0	1	2	3	4	5	6	7	8	9	10	11	12
$\mathcal{T}$	d	b	d	b	**c**	**a**	**b**	b	a	d	a	d	b
$\mathcal{P}$				a	**c**	**a**	**b**						
shift								b	c	a	b		

Table 2.7: *In this case, a mismatch occurs comparing b-a. So we have to shift the pattern to align it to the rightmost occurrence of b in $\mathcal{P}$. Since there is any, we simply move the pattern past the character a*

character that only occurs at the last index, it will be mapped to $|p|$. The init function of BMH algorithm can be done as in program 2.5.

Program 2.5 BMH init function.

```
FUNCTION BMHinit(𝒜, p)
 ▹ BMHtable[s] stores the righmost occurence of S in p[0 : m − 2]
    for s ∈ 𝒜
        BMHtable[s] = |p|
    end
    i ⟵ 0
    for c ∈ p
        BMHtable[c] ⟵ i
        i ⟵ i + 1
    end
end
```

2.4.2 Search

The search phase is as in the Boyer-Moore algorihtm. We compare the pattern from right to left with the text and we shift the pattern according to the BMHtable if a mismatch occurs. Check program 2.6

Program 2.6 BMH search function.

```
FUNCTION BMHsearch(BMHTable)
    i ⟵ 0
    while i ≤ |T| − |P|
        while j ≥ 0 and P[j] == T[i + j]
            j ⟵ j − 1
        if j < 0
            ans ⟵ ans ∪ {i}
        i ⟵ i + |P| − 1
        i ⟵ i − BMHtable[T[i]]
        end
    end
return ans
```

2.4.3 Example 1

Let $p := ACTG$ be a pattern and $T := AAACCTAGACTGA$ be a text. Let's build the Bad-Character Table for the characters of p, as we can see in 2.8

c	value
A	$4 - 0 - 1 = 3$
C	$4 - 1 - 1 = 2$
T	$4 - 2 - 1 = 1$
G	4 (last character)
*	4

Table 2.8: *Bad-Character table for $p := ACTG$*

Now, let's use the BMH Algorithm to find all the ocurrences of p in $\mathcal{T}$

At iteration i = 0, we compare the character G with C, and since $G! = C$ we shift the pattern by $BMHTable[G] = 4$.

$$AAA\underline{C}CTAGACTGA$$
$$ACT\underline{G}$$

At iteration i = 1, we compare the character G with G, since it matches we proceed to compare T with A. Since it doesn't match, we shift the pattern by $BMHTable[G] = 4$.

$$AAACCT\underline{A}\mathbf{G}ACTGA$$
$$AC\underline{T}\mathbf{G}$$

At iteration i = 2, finally all characters of the pattern have matched. We have found a match at index 8. We shift the pattern by 4 (its size), which ends the algorithm.

$$AAACCTAG\mathbf{ACTG}A$$
$$\mathbf{ACTG}$$

2.4.4 Example 2

Let $p := abra$ be a pattern and $T := abracadabra$ be a text. Let's build the Bad-Character Table for the characters of p, as we can see in table 2.9

At iteration i = 0, we find a match and shift the pattern by 1.

c	value
a	$4 - 0 - 1 = 3$
b	$4 - 1 - 1 = 2$
r	$4 - 2 - 1 = 1$
*	4

Table 2.9: *Bad-Character table for* $p := abra$

abra$cadabra$
abra

At iteration i = 1, we compare the characters c with a, which doesn't match. We shift the pattern by BMHTable[a] = 3

$abra\underline{c}adabra$
$abr\underline{a}$

At iteration i = 2, we compare the characters a with a, which matchs, and proceeds to compare d with r. Since it doesn't match we shift the pattern by BMHTable[a] = 3.

$abraca\underline{d}$**a**bra
$ab\underline{r}$**a**

At iteration i = 3, we find another match. We shift the pattern by 1, which ends the algorithm.

$abracad$**abra**
abra

2.4.5 Complexity

The time complexity of BMHinit is given by the loops on lines 3 and 7, so its time complexity is $O(\mathcal{A} + |P|)$ and the space complexity is the size of the BMHtable, which is $O(|A|)$.

The BMHsearch does not improve the performance of heuristics from BMsearch and it is just a simplified version. The time complexity of BMHsearch is given by the number of iterations on line 5. Like in BMsearch, the worst case is $\mathcal{O}(|\mathcal{T}||\mathcal{P}|)$ and the best case is when the first comparison on line 4 holds false for every character and then the algorithm performs just $O(|T|/|P|)$ comparisons. The space complexity of BMHsearch is $\mathcal{O}(1)$, since we do not need any additional memory beside loop variables.

2.5 KMP Algorithm

The Knuth-Morris-Pratt Algorithm Knuth *et al.*, 1977 solves the SPSM problem by constructing an auxiliary function called Prefix Function in order to find the occurrences of the pattern without performing any strings comparisons.

2.5.1 Prefix Function

Let s be a string of length n. For each integer $i \in \{0 \cdots n-1\}$ define $m := s[0 \cdots i]$. The Prefix function for s is an array π such that $\pi[i]$ is the length of the longest proper prefix of m which is also a suffix of m ending at index i.

That is,

$$\pi[i] = max_{k=0 \cdots i}\{k : s[0 \cdots k-1] = s[i-(k-1) \cdots i]\}$$

2.5.2 Example

The prefix function for the string "abracadabra" is $[0, 0, 0, 1, 0, 1, 0, 1, 2, 3, 4]$.

2.5.3 Proposition 1

Let s be a string of length $n > 1$. For each integer $i \in \{0 \cdots n-2\}$ we have $\pi[i+1] \leq \pi[i] + 1$.

Proof. For the sake of contradiction, suppose there is an integer i such that $\pi[i+1] > \pi[i]+1$. Then, there is a suffix of $m := s[0 \cdots i+1]$ of length $\pi[i+1]$, we remove the last character of m yielding the string $m' := s[0 \cdots i]$ of length $\pi[i+1] - 1$, which, by hypothesis, is greater than $\pi[i]$. This contradicts the definition of $\pi[i]$. □

2.5.4 Proposition 2

If $i > 0, s[i+1] = s[\pi[i]]$, then $\pi[i+1] = \pi[i] + 1$

Proof. By definition of π, we now the prefix that starts at index i has largest border $b := s[0 \cdots \pi[i] - 1]$ with length $\pi[i]$. The suffix for b is $s[i - \pi[i] + 1 \cdots i]$. So, s[pi[i]] is the next character after b and $s[i+1]$ is the next character of the suffix of b. Therefore, if $s[i+1] = s[\pi[i]]$, then $\pi[i+1] = \pi[i] + 1$. □

2.5.5 Optimizations

With proposition 2, we can calculate $\pi[i+1]$ from $\pi[i]$ if $s[i+1] = s[\pi[i]]$. If that is not the case, then we find the largest $j < \pi[i]$ such that the prefix property holds for j and check if $s[i+1] = s[j]$. Notice that such j is in fact $\pi[i-1]$. Therefore, if $s[i+1] = s[\pi[i-1]]$, we assign $\pi[i+1] = \pi[\pi[i-1]] + 1$, otherwise we set $j := \pi[i-1]$ and repeat the process.

2.5.6 Preprocessing

With the optimizations, we can construct the final algorithm. Check Program 2.7.

Program 2.7 KMP Algorithm prefix function

```
FUNCTION KMPPrefix(p)
    i ⟵ 0
    π[0] = 0
    for i in {1 ··· |p| − 1}
        j ⟵ π[i − 1]
        while j > 0 and s[i] ≠ s[j]
            j ⟵ π[j − 1]
        end while
        if s[i] = s[j]
            j ⟵ j+1
        π[i] = j
    end for

return π
```

2.5.7 Search

So, how do we perform searches? Let T be a text and p a pattern. Define $s := p + * + T$, where $*$ is an arbitrary character such that there is no occurrence of in T neither in p. We build the prefix function for s. Then, if there is an integer $i > n$ such that $\pi[i] = n$, then there is an occurrence of p in T at the index $i - 2n$.

Proof. By the definition of s, if $i > n$, then $s[i]$ is a character of $\mathcal{T}$. By the definition of pi, if $\pi[i] = n$, then there the prefix of s of size n coincides with $s[i - (n + 1) - n + 1 = i - 2n$, but a prefix of s of size n is p, by construction. Therefore, there is an occurrence of p in $\mathcal{T}$ at the index $i - 2n$. □

The program 2.8 shows the search phase of the KMP Algorithm

Program 2.8 KMP Search

```
FUNCTION KMPPrefix(p, T, π)
    z := p + * + t
    for value ∈ π do
        if value = |p|
            match found
    end for
end
```

2.5.8 Complexity

From proposition 1, the prefix function, at each iteration, can increase by at most 1. Therefore, the While loop in line 7 of program 2.7 can perform at most n iterations at total since it is limited by the value of π. Thus, the program 2.7 has $\mathcal{O}(|\mathcal{T}|)$ of time complexity and $\mathcal{O}(|\mathcal{T}|)$ of space complexity. The Search phase (program 2.8) is $\mathcal{O}(|T|)$ of time complexity and $\mathcal{O}(1)$ of space complexity.

Therefore, the KMP algorithm is $\mathcal{O}(|\mathcal{T}|)$ of time complexity and $\mathcal{O}(|\mathcal{T}|)$ of space complexity.

2.5.9 Example 1

Let $\mathcal{T} := DACDACAC$ and $p := ACAC$. Let's use the KMP Algorithm to find all occurrences of p in $\mathcal{T}$. First, let's construct the prefix function for $\mathcal{T} + {}^* + p$ which is ACAC*DACDACAC.

For i=0, $s[0 \cdots i] = A$. We have no borders. So, $\pi[0] = 0$

For 1=1, $s[0 \cdots i] = AC$. Again, there are no borders. So, $\pi[1] = 0$

For 1=2, $s[0 \cdots i] = ACA$. We have border A. So, $\pi[2] = 1$

For 1=3, $s[0 \cdots i] = ACAC$. We have borders A and AC. So, $\pi[3] = 2$

For 1=4, $s[0 \cdots i] = ACAC*$. We have no borders. So, $\pi[4] = 0$

For 1=5, $s[0 \cdots i] = ACAC * D$. We have no borders. So, $\pi[5] = 0$

For 1=6, $s[0 \cdots i] = ACAC * DA$. We have border A. So, $\pi[6] = 1$

For 1=7, $s[0 \cdots i] = ACAC * DAC$. We have borders A and AC. So, $\pi[7] = 2$

For 1=8, $s[0 \cdots i] = ACAC * DACD$. We have no borders. So, $\pi[8] = 0$

For 1=9, $s[0 \cdots i] = ACAC * DACDA$. We have border A. so $\pi[9] = 1$

For 1=10, $s[0 \cdots i] = ACAC * DACDAC$. We have borders A and AC. So, $\pi[10] = 2$

For 1=11, $s[0 \cdots i] = ACAC * DACDACA$. We have borders A, AC and ACA. So, $\pi[11] = 3$

For 1=12, $s[0 \cdots i] = ACAC * DACDACAC$. We have borders A, AC and ACAC. So, $\pi[12] = 4$

Thus, the Table 2.10 shows the prefix function for $\mathcal{T} := DACDACAC$ and $p := ACAC$.

Now, for the searching phase we just have to iterate over the values of π and check if it's equal to $|p| = 4$. In this case, at $i = 12$ we have found a match at index $i - 2 * |p| = 12 - 8 = 4$ of $\mathcal{T}$.

i	$\pi[i]$
0	0
1	0
2	1
3	2
4	0
5	0
6	0
7	0
8	0
9	1
10	2
11	3
12	4

Table 2.10: *Prefix function for* $\mathcal{T} := DACDACAC$ *and* $p := ACAC$.

2.6 Shift-Or Algorithm

The Shift-Or Algorithm Baeza-Yates and Gonnet, 1989 uses bitwises techniques to solve the Pattern Matching Problem. Using the same principles of the previous algorithms, we have a preprocessing phase and a search phase. However, we don't perform any string comparisons, instead the algorithm will be based on the Shift and Or bitwise operators, which explains its name.

2.6.1 Bitmask

Let $p := p_0 \cdots p_n$ be a pattern of size n. The Shift-Or Algorithm constructs a hash table mask that maps each character c of p to a bitmask $d = d_n \cdots d_1$, where $d_i = \{0, 1\}$. mask[c] has I-th bit set to 0 if, and only if, $p_i = c$. For instance, for the pattern $abra$, we have the bitmask table as in 2.11

character	d_1	d_2	d_3	d_4	bitmask ($d_4 \cdots d_1$)
a	0	1	1	0	0110
b	1	0	1	1	1101
r	1	1	0	1	1011
*	1	1	1	1	1111

Table 2.11: *Bitmask table for* $p := abra$

2.6.2 Matching

With the table mask created, we can find the occurrences of the pattern p in the text T with the following algorithm:

1. We start with a bitmask ϕ with $\neg 1$ i.e, the bitwise negation of 1 (all bits set to 1 except for 2^0).

2. For each character c of $\mathcal{T}$, we perform the OR bitwise operator with ϕ and $mask[c]$ and shift phi to the left. Thus, we apply the assignment:

$$(\phi|mask[c]) << 1$$

3. If at any step, the d_m bit is set to 1, then a match was found at index $i - n + 1$.

2.6.3 Algorithm

We are going to combine the preprocessing phase with the matching phase to build an algorithm on the fly. To do so, we start constructing the ϕ table, and if at any moment, we have found that the d_m bit in mask is set to 0, the we have found a match.

The program 2.9 shows the Shift-Or Algorithm.

Program 2.9 Shift-Or Algorithm

```
FUNCTION shiftOr(p, T)
 ▹ mask table is initialized with mask with all bits set to 1
 ▹ φ is initialized with mask with all bits set to 1
    i ⟵ 0
    while i < |p| do
        c ⟵ p[i]
        mask[c] = mask[c] & −(2 ∗ 10^i)
        i ⟵ i + 1
    end while
    for c ∈ 𝒯 do
        φ ⟵ R | mask[c]
        φ ⟵ R << 1
        if (R & (2 ∗ 10^m)) = 0
            match found
    end for
end
```

2.6.4 Example 1

Let $p = abra$ be a pattern and $T = abracadabra$ a text. Let's use the Shift-Or algorithm to find the occurrences of p in $\mathcal{T}$. For this, we use the mask table 2.11

We start with $\phi = 11110$.

Then, we have the execution trace of the algorithm in table 2.12.

As we can see, we have found matches at iteration 3 and 10.

i	ϕ	c	$mask[c]$	$\phi \mid mask[c]$	$(\phi \mid mask[c]) << 1)$
0	11110	a	10110	11110	11100
1	11100	b	11101	11101	11010
2	11010	r	11011	11011	10110
3	10110	a	10110	10110	01100
4	01100	c	11111	11111	11110
5	11110	a	10110	11110	11100
6	11100	d	11111	11111	11110
7	11110	a	10110	11100	11100
8	11100	b	11101	11101	11010
9	11010	r	11011	11011	10110
10	10110	a	10110	10110	01100

Table 2.12: *Execution trace of the Shift-Or Algorithm for* $p = abra$ *and* $T = abracadabra$

2.6.5 Example 2

Let $p = ACTG$ be a pattern and $T = ATAACTGTCA$ a text. Let's use the Shift-Or algorithm to find the occurrences of p in $\mathcal{T}$.

First, we generate the bitmask table, as we can see in table 2.13.

c	$b_1 b_2 b_3 b_4$	$bitmask[c]$
A	0111	1110
C	1011	1100
T	1101	1011
G	1110	0111
*	1111	1111

Table 2.13: *Bitmask table for* $p = ACTG$

Next, starting with $\phi = 11110$, the table 2.14 shows the execution trace of the Shift-Or Algorithm for this input.

As we can see, we have found a match at index 3 at iteration 6.

2.7 Trie Data Structure

The Trie Data Structure is one of the most useful data structure for solving string problems. It allows us to perform pattern matching by constructing a tree of characters nodes. Moreover, the Aho-Corasick Algorithm is fundamentally based on the Trie Data Structure. So, we must study it first.

A trie is a tree of characters that encode a set of strings. A path from root to a node in the tree represents a string. The figure 2.1 shows the Trie Data Structure for the set of strings $\mathcal{S} = \{ada, abc, cba, add, abra\}$. Notice that the root is a special node, and it is not represented by any character. Also, for every path from root to a node colored in red, the

i	ϕ	c	$mask[c]$	$\phi \vert mask[c]$	$(\phi \vert mask[c]) << 1)$
0	11110	A	11110	11110	11100
1	11100	T	11011	11111	11110
2	11110	A	11110	11110	11100
3	11100	A	11110	11110	11100
4	11100	C	11101	11101	11010
5	11010	T	11011	11011	10110
6	10110	G	10111	10111	01110
7	01110	T	11011	11111	11110
8	11110	C	11101	11111	11110
9	11110	A	11110	11110	11100

Table 2.14: *Execution trace of the Shift-Or Algorithm for* $p = ACTG$ *be a pattern and* $T = ATAACTGTCA$

concatenation of the characters represented in the nodes will result in a string of S. The nodes colored in red are called "word nodes", and they are special because they represent ending characters of a string in $\mathcal{S}$. The purpose of this shall be clarified when we discuss how to perform searches.

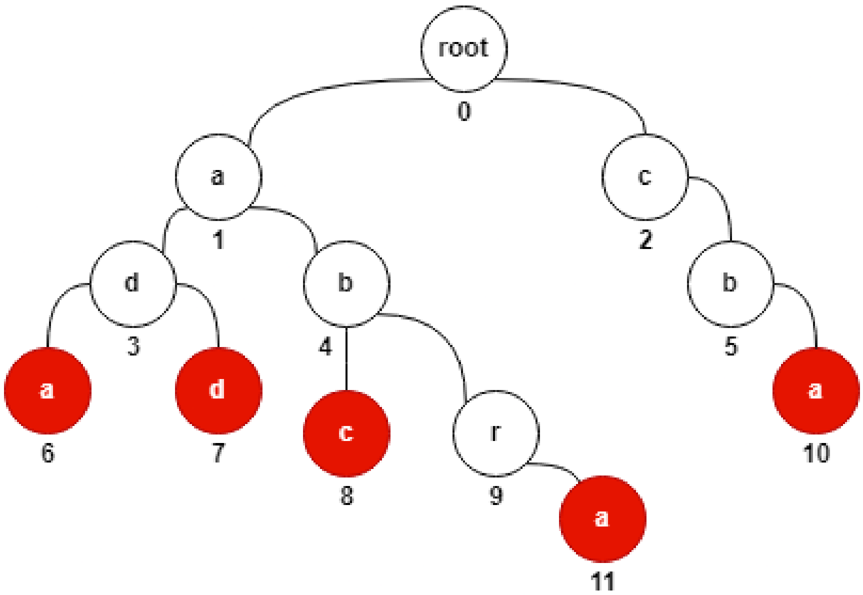

Figure 2.1: *Trie Data Structure for the set of strings* $\mathcal{S} = \{ada, abc, cba, add, abra\}$.

The trie node can be defined as:

Program 2.10 Trie node

```
struct trieNode
    children ← (c ∈ 𝒜 → TrieNode)   ▷ Character map function
    word ← false
```

2.7.1 Construction

To construct the Trie Data Structure from the set of patterns, we start with the empty trie, which is the one with only the root node, and insert each string of P into the trie. Therefore, the construction is simply a loop of insertions. Now, let's describe how to insert a string s in the trie.

We are going to insert each character of s separately. Starting from the root node, we check if there is already a child representing the first character of s. Is so, then we simply move to this node. Otherwise we create a new node and move to it. After moving to the next node, we also proceed to the next character in s and repeat the process. After adding the last character, we mark its representative node as a word node.

Example

In the figure 2.1, we have a trie for the set of strings $\mathcal{S} = \{ada, abc, cba, add, abra\}$. Now let us add the string abbd. We start at the root (node 0) and considering the character a. Since there is already a node from node 0 to a node with label a, the node 1, so we move to node 1 start considering the next character in abbd, which is b.
Again, there is a child of node 1 with label b, the node 4. So we move to node 4 and start considering the next character in abbd, which is b.
From node 4, there is no child with label b, so we create a new node, namely node 12, and move to it. We now consider the next character in abbd, which is d.
From node 12, there is no child with label d. So we create a new node, namely node 13, and move to it. Since this is the last character of abbd, we mark this node as a word node.

Here, the code for inserting a string in the trie is given:

Program 2.11 Trie insert function.

```
FUNCTION trieInsert(root, s)
    pointer ← root
    for c ∈ s
        if root.children[c] is not defined
            root.children[c] ← TrieNode(c)
        pointer ← root.children[c]
    end
    pointer.word ← true
return ans
```

2.7.2 Pattern Searching

We can use the Trie Data Structure to perform pattern searches. The idea is to build a trie of all suffixes of the text $\mathcal{T}$ and use it to check if there is a occurence of a pattern p in T. We process each character c in the pattern p and, starting with the root, we follow the edges of the trie for each character. if, at any moment, there is no such edge, then the pattern doesn't occur in $\mathcal{T}$. If we find a leaf, we have found a match.

Example 1

Let $\mathcal{T}$ = abracadabra, and $p = abra$. First, we generate all suffixes of T and build a trie with them. The table shows all suffixes of T, we use \$ to denote the ending of T.

i	suffix
0	abracadabra$
1	bracadabra$
2	racadabra$
3	acadabra$
4	cadabra$
5	adabra$
6	dabra$
7	abra$
8	abra$
9	bra$
10	ra$
11	a$
12	$

Table 2.15: *The suffixes of abracadabra*

Then, we build a trie with all suffixes of $\mathcal{T}$, as we can see in the figure 2.2

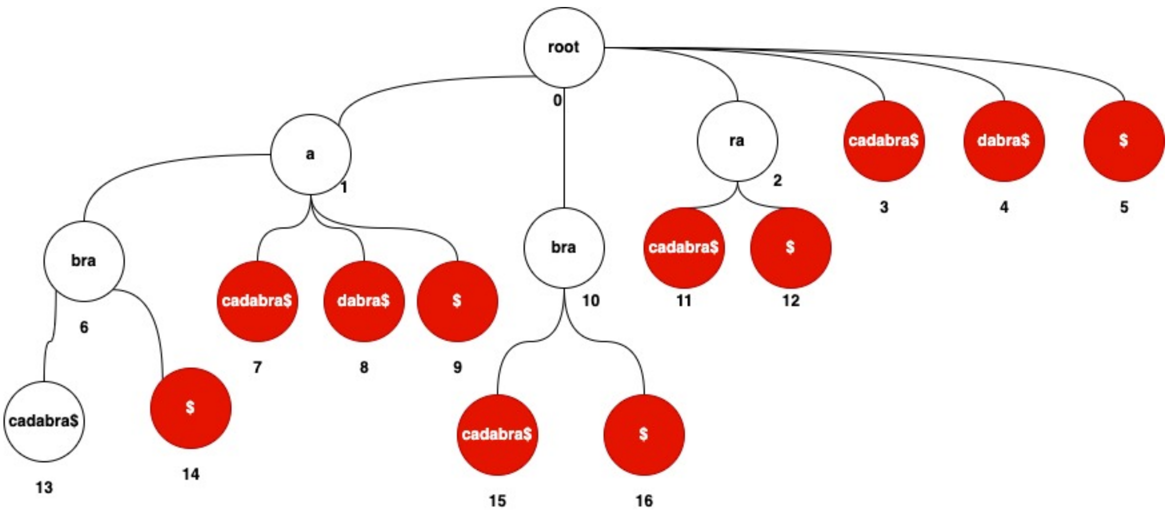

Figure 2.2: *Trie of suffixes for* $\mathcal{T}$ = *abracadabra*

Now, we are going to iterate over each character of the pattern p while we move through the Trie. Starting at the root, we check if there is an edge with label corresponding to the next character in p. If so, then we follow this edge and repeat. If, at any moment there is no such edge, we conclude that the pattern does not exist in T and return. If all characters have been processed, then we print the suffix list of the current node, which yields the occurences of p in T.

We start at the root. The next character in p is 'a', so we go to node 1.

The next characters are 'b', 'r' and 'a', we go to node 6.

Finally, the next character is \$, we go to node 14. Since this is a word node, we have found a match and we print the list of indices stored in this vertex.

Example 2

Let $p = abc$ be a pattern and $\mathcal{T} = ababbabcbaabcc$ be a text. Let us generate all suffixes of T, as we can see in table XX.

i	suffix
0	ababbabcbaabcc\$
1	babbabcbaabcc\$
2	abbabcbaabcc\$
3	bbabcbaabcc\$
4	babcbaabcc\$
5	abcbaabcc\$
6	bcbaabcc\$
7	cbaabcc\$
8	baabcc\$
9	aabcc\$
10	abcc\$
11	bcc\$
12	cc\$
13	c\$
12	\$

Table 2.16: *The suffixes of ababbabcbaabcc*

Then, we build a Trie of all suffixes of $\mathcal{T}$, see figure 2.3

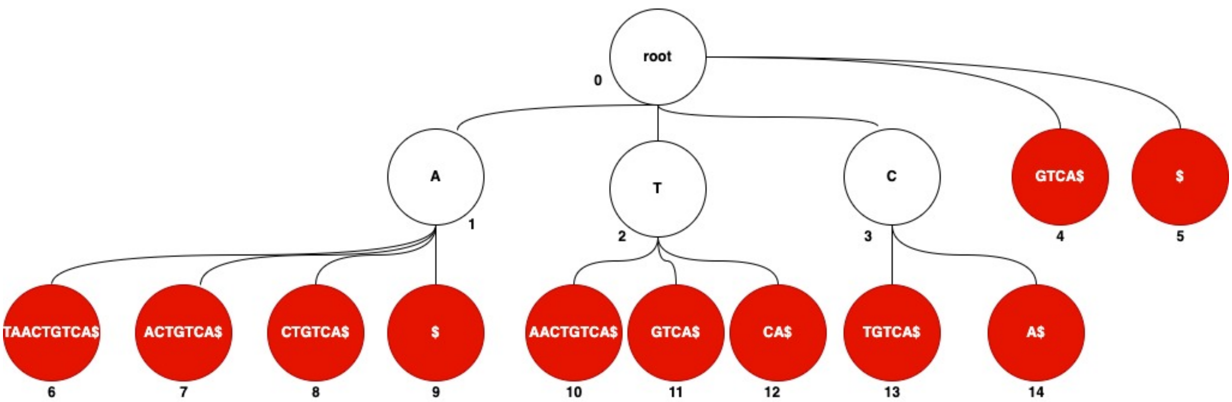

Figure 2.3: *Trie of suffixes for* $\mathcal{T} = ababbabcbaabcc$

Finally, we process each character of p and follow the edges of the Trie.

Complexity

The complexity of this algorithm is given by the preprocessing phase and the searching phase. We can build the Trie in $\mathcal{O}(|\mathcal{T}|^2)$ by generating all suffixes of $\mathcal{T}$. And then the search phase is simply iterating over each character of P and following edges, which can be done in $\mathcal{O}(|P|)$.

Chapter 3

Multiple Pattern String Matching

3.1 Introduction

In this chapter, we discuss algorithms to solve the following problem: Given a text $\mathcal{T}$ and a set of patterns $\mathcal{P}$, where all symbols are from an alphabet $\mathcal{A}$, find all occurrences of all patterns of $\mathcal{P}$ in $\mathcal{T}$. The algorithms we are going to study in this chapter have several similarities with the previous algorithms. Thus, it's very important to understand those algorithms first.

3.2 Aho-Corasick

The Aho-Corasick algorithm Aho and Corasick, 1975 is an extension of the Trie Data Structure to deal with multiple patterns. We construct an automaton by constructing a Trie from the set of patterns and then adding some addional links, the suffix link and the exit link.

3.2.1 Preprocessing

The preprocessing of the Aho-Corasick is quite different from the other algorithms. We are going to describe how to construct a **finite deterministic automaton** from the Trie Data Structure. Suppose we have a set of patterns $\mathcal{P}$ and a trie $\mathcal{T}$ for the set of strings $\mathcal{P}$. We construct an automaton in which every vertice v of $\mathcal{T}$ is a state, and for every edge e of the trie, we have a transition according to the corresponding letter. Notice that this does not define an automaton yet, since we have to define a transition from every letter from the alphabet. If there is no corresponding edge in the trie, then we have to go into some state. Let us describe formally how the Aho-Corasick deals with this problem.

Suppose we are currently in some state p, obtained from some vertice v in the trie. Define the string $s[v]$ corresponding to v as the string obtained by the character nodes from the path from the root to the vertice v. Suppose we are processing a character c, if there is an edge in the Trie from v, with label c, then we can go over this edge and get the corresponding vertex and new state. Otherwise, we find the state corresponding to the longest proper suffix of the string $s[v]$, and try to perform the transition from there.

3.2.2 Suffix Link

A **suffix link** is an edge that leads to the proper suffix of the string $s[v]$. We build such links recursively. The base case is the root of the trie, in which the suffix link will point to itself. The general case we have some vertex v, and there is no transition from v, with letter c. Then, we can go to the ancestor p of v, follow its suffix link, which is already defined by induction, leading to some vertex u, and try to perform the transition with the letter c from u. If there is no such edge, we repeat until we reach the root, our base case. Therefore, we can build such links in linear time proportional to the height of the Trie.

Example

Let $\mathcal{T} = abcabda$ and $\mathcal{P} = [bc, bd, abc, abd]$. Let's construct the Aho-Corasick data structure. Let's add the strings bc, bd, abc and abd to the Trie and add its suffix links (See figure 3.1). The dashed arrows represents suffix links and the red nodes represents word nodes.

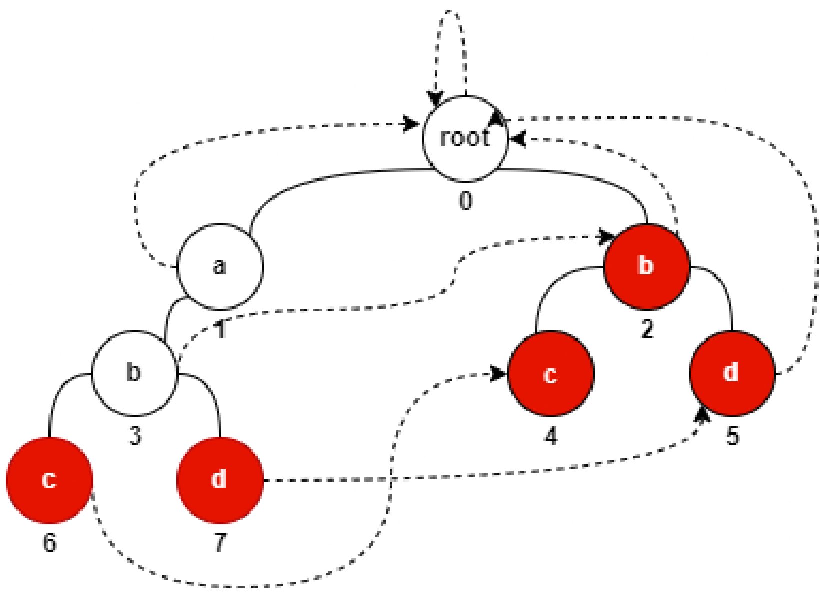

Figure 3.1: *Aho Corasick Data Structure for* $\mathcal{T} = abcabda$ *and* $\mathcal{P} = [bc, bd, abc, abd]$.

The node 1 represents the string "a". Since there are any proper suffixes of "a", we just add a suffix link pointing to the root.
The node 2, represents the string "b". Again, there are any proper suffixes, so we add a suffix link pointing to the root.
The node 3 represents the string "ab", the only proper suffix of "ab" is "b", which exists in the trie (node 2). So we add a suffix link pointing to node 2.
The node 4 represents the string "bc", the only proper suffix of "bc" is "c", which doesn't

exists in the trie. So we just a suffix link pointing to the root. Since "bc" is a pattern, we mark this node as a word node.
The node 5 represents the string "bd", its only proper suffix is "d", which doesn't exists in the trie. So we add a suffix link pointing to the root.
The node 6 represents the string "abc", its proper suffixes are "c", and "bc". The longest one is "bc", which exists in the Trie (node 4). So we add a suffix link pointing to node 4. Since "abc" is a pattern, we mark this node as a word node.
The node 7 represents the string "abd", its proper suffixes are "d" and "bd". The longest one is "bd", which exists in the Trie (node 5). So we add a suffix link pointing to node 5. Since "abd" is a pattern, we mark this node as a word node.

3.2.3 Search

So, how do perform searches? We iterate over the text $\mathcal{T}$ and we perform the transitions for every letter, starting from the root. If the next letter exists in the Trie, we simply go over its edge. Otherwise we must follow the suffix link and try again. Whenever we pass through a word node, we print the corresponding match.

Example

In the previous example, $\mathcal{T} = abcabda$ and $\mathcal{P} = [bc, bd, abc, abd]$. We start at the root, and try to perform the transition to letter 'a', moving to node 1.
The next letter is 'b', so we move to node 3. The next letter is 'c', so we move to node 6. Node 6 is word node, so we print its corresponding match which is "abc".
The next letter is 'a'. There is no edge from the node 6 to a node with the letter 'a', so we go over its suffix link and move to node 4 and try to perform the transition from there. Node 4 is a word node, so we print the pattern "bc". However, again there is no edge to a character 'a', so we go over its suffix link and move to node 0 (the root) and perform the transition to node 1.
The next letter is 'b', so we move to node 3. The next letter is 'd', so we move to node 7. Node 7 is a word node, so we print the pattern "abd".
The next letter is 'a'. Since there is no transition to a node with letter 'a' from the node 7, we go over its suffix link (which points to the root) and perform the transition to node 1, which ends the search.

3.2.4 Exit Links

How do we verify the matches with this automaton? It is clear that whenever we reach a leaf vertex v, then the string $s[v]$ is a match. But, there may be one, or several other matches. If we reach a leaf vertex and move along the suffix links, then there will be a match for every leaf that we find. To speed up this process of finding new matches, the Aho-Corasick also creates another type of link: the **exit link**, which is simply the nearest leaft vertex that is reachable using suffix links. Again, we can construct such links in a recursive way.

Example

Let $\mathcal{T} = abcd$ and $\mathcal{P} = [ab, abc, abcde, d]$. The next figure shows the Aho-Corasick data structure with just the suffix links.

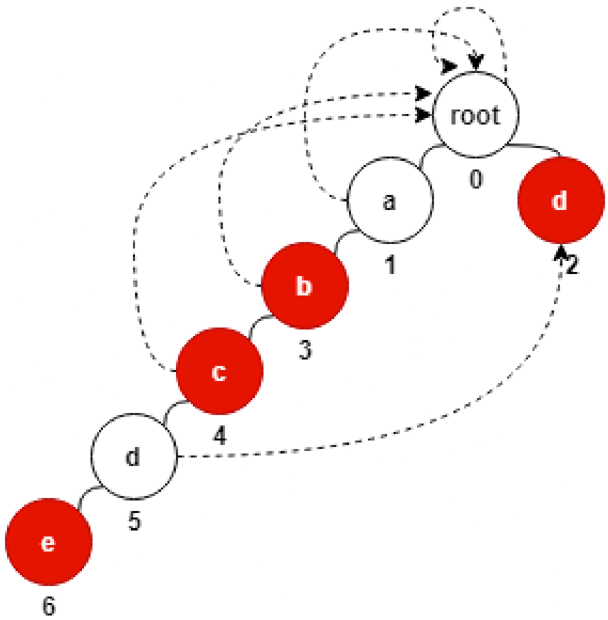

Figure 3.2: *Aho Corasick Data Structure for* $\mathcal{T} = abcd$ *and* $\mathcal{P} = [ab, abc, abcde, d]$.

Now let us use the automaton to find matches. We start at root (node 0). The first letter in $\mathcal{T}$ is 'a'. So we move to node 1. The next character is 'b', so we move to node 3. Since node 3 is a word node, we print the match "ab". The next character is 'c', so we move to node 4. Again, the node 4 is a word node, so we print the match "abc". The last character is 'd', so we move to node 5 and we finish our search. However, the pattern 'd' was not found. We can fix this problem by adding the exit links.

The next figure shows the data structure with the exit links added. Now, for every character, we must check its exit link, if it points to a leaf node, then we have found a match! In the previous case, the node 5 now has a exit link to node 2, which is a word node and the pattern "d" can now be found.

3.2.5 Final Algorithm

As some of the previous algorithms, the Aho-Corasick Algorithm is also composed of two phases: the build phase and the search phase. During the build phase, we build the suffix and exit links, which will be implemented using arrays. The build phase can be decomposed in two sub-phases:

- We build the trie by adding all characters of all patterns. In this step, we build the array go, which will be used to determine the next state in the automaton, and start building the array exit, which will represent a map from a state to a bitmask representing the exit links.

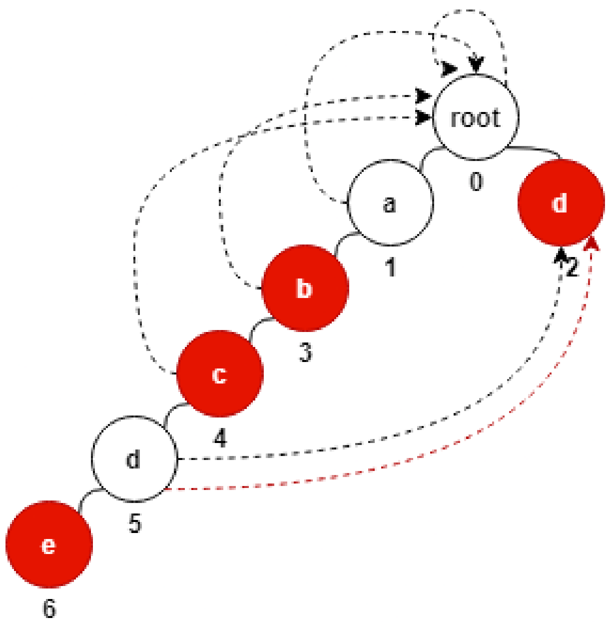

Figure 3.3: *Aho Corasick Data Structure, now with exit links added, for* $\mathcal{T}$ = *abcd and* $\mathcal{P}$ = $[ab, abc, abcde, d]$.

- We then use a queue data structure to build the suffix links and finishing building the exit links.

The algorithm 3.1 shows the build phase of the Aho-Corasick algorithm.

Program 3.1 Aho-Corasick build phase

```
FUNCTION build(𝒫)
  ▻ exit[state] will map state to a bitmask representing the index of patterns reachable using
    exit links
  ▻ go[state][c] will represent the new state obtained from state following the character c edge
  ▻ suffix[state] will represent the new state obtained from state following the suffix link
    count ⟵ 0   ▻ Will be used to track the index of patterns
    for p ∈ 𝒫
        currentState ⟵ 0
        for c ∈ p
            if go[currentState][c] is ∅
                go[currentState][c] ⟵ new state
            currentState ⟵ go[currentState][c]
        end
  ▻ Here we set the bit representing the pattern p in the final state from previous loop
        exit[currentState] |= (1 << count)
        count ⟵ count + 1
    end
return exit, go, suffix
```

Finally, the algorithm 3.2 shows the search phase of the Aho-Corasick algorithm. Here, we use an auxiliary function called next, which will be used whenever we are have found no transition from a particular state with letter c and we have to traverse the suffix links to keep trying.

Program 3.2 Aho-Corasick Search Phase

```
FUNCTION next(go, currentState, c)
    while go[currentState][c] is ∅
        currentState = suffix[currentState]
    end
    return go[currentState][c]

FUNCTION search(𝒯, 𝒫, exit, go, suffix)
    currentState ⟵ 0
    for c ∈ 𝒯
        currentState ⟵ next(currentState, c)

 ▹ One or several matches found.
        if out[currentState] is not ∅
            count ⟵ 0
            for p ∈ 𝒫
                if (out[currentState] & (1<<count)
                    match found
                count ⟵ count + 1
            end
    end
```

3.2.6 Complexity

The build phase of the Aho-Corasick algorithm is $\mathcal{O}(|\mathcal{P}|)$ of time-complexity. Now, for the search phase, we process each character c of $\mathcal{T}$ and we have to check for matches. Therefore, the time-complexity of the Aho-Corasick algorithm is $\mathcal{O}(\mathcal{T} + |ans|)$, where $|ans|$ is the number of matches found. For memory, we use $\mathcal{O}(|\mathcal{A}||\mathcal{P}|)$, where $|\mathcal{A}|$ is the size of the alphabet.

3.3 Wu-Manber

The Wu-Manber algorithm Wu and Manber, 1994 is mainly an extension from the Boyer-Moore algorithm to deal with multiple patterns. As with the other algorithms, we have a preprocessing phase of building some tables and then a search phase. We are going to build three tables, a SHIFT table, similar to the Boyer-Moore shift table, and a HASH table and PREFIX tables, which uses some heuristics to check matches.

3.3.1 Polynomial rolling hash

Before we study the Wu-Manber algorithm, we present a string hash technique which is called the polynomial rolling hash of a string s. This hash technique will be used in the Wu-Manber algorithm.

Let p and m be fixed positive numbers, the Polynomial Rolling Hash of a string s is

$$hash(s) = s[0] + s[1] * p + s[2] * p^2 + \cdots + s[n-1] * p^{n-1} \mod m$$

3.3.2 Preprocessing

Suppose we have a set of patterns $\mathcal{P}$. Let m be the minimum length of a pattern, i.e m is the minimum size $|p|$ for all $p \in \mathcal{P}$. Also, let M be the total size of the patterns and c the size of the alphabet. We are going to consider the first m characters of each pattern in order to build the tables. Also, we are going to consider a sliding window in the text T, of size B, where $B = log_c 2M$. To build the SHIFT table, we consider each pattern $p \in \mathcal{P}$ and each substring s of size B of p. $SHIFT[s]$ will be the largest possible value for a shift. Let T_i the text window of size B obtained at iteration i, i.e, $T_i = T_i \cdots T_{i+B-1}$. We have two cases:

- T_i occurs in some pattern, i.e, T_i is a substring of one or more patterns in $\mathcal{P}$
- T_i does not occur in any pattern $p \in \mathcal{P}$

If case 1 holds, then we have to shift the sliding window to allow for all occurrences of T_i be checked. Therefore, similar to the Boyer-Moore algorithm, we find the rightmost occurrence of T_i in P, which occurs in some index q. And we set $SHIFT[i] \longleftarrow m - q$ (see figure 3). Suppose case 2 holds, then we can shift our sliding window by $m - B + 1$ characters, since any smaller shift would get a mismatch. We can set $SHIFT[i] \longleftarrow m - B + 1$.

The program 3.4 shows the Preprocessing phase of Wu-Manber algorithm

Program 3.3 Wu-Manber Preprocessing

```
FUNCTION hash(s)
    p ⟵ 31
    mod ⟵ 1 * 10^9 + 9
    hash ⟵ 0
    power ⟵ 1
    for c ∈ s:
        hash ⟵ (hash + value(c) * power) % mod
        power ⟵ (power * p) % mod
    return hash

FUNCTION preprocess(𝒫, B)
    m ⟵ inf

    for p ∈ 𝒫 do
        m ⟵ min(|p|, m)
    end for

    # SHIFT table is initialized with m − B + 1
    for p ∈ 𝒫 do
```

cont ⟶

```
⟶ cont
        j ← m
        while j ≥ B do
            hash ← compute_hash(p[j−B ··· j])
            shift ← m − j
            SHIFT[hash] ← min(SHIFT[hash], shift)
            if shift = 0
                prefixHash = compute_hash(p[0 ··· 1]
                idx = i
                push(TABLE[hash], (prefixHash, idx))
        end while
    end for
    return SHIFT, TABLE
```

Example

Let $\mathcal{T} = abracadabra$ and $\mathcal{P} = [abra, cada, bra, aca]$. Suppose $B = 2$. In this case, $m = 3$, and considering only the first m characters of each pattern we get $\mathcal{P}' = [abr, cad, bra, aca]$. The prefix table will contain the polynomial rolling hash for each prefix of size 2 (because we set $B = 2$). The table 3.1 shows the prefix table for this example.

Prefix Table	
ab	63
ca	34
br	560
ac	94

Table 3.1: *Prefix Table for* $\mathcal{T} = abracadabra$ *and* $\mathcal{P} = [abra, cada, bra, aca]$ *with* $B = 2$

Now, for each pattern in P', we consider its suffix of size 2 to construct the hash table. We have the suffixes "ca", "ra", "ad" and "br". We map each suffix to a list of pairs. The first value is the pattern in P' which has the key as suffix, and the second value is the polynomial rolling hash for the prefix of this pattern.

For the shift table, the initial value of each key will be $m - B + 1$. We map each substring s of size B from P' to the minimum of its current value and $m - j$, where j is the position of the last character of s in $P_i = a_1 \cdots a_{|P_i|}$, $P_i \in P'$. Let's see how it's done in this example.

We have the substrings "ab", "br", "ca", "ad", "ra", and "ac", and the initial value is 2, since $m = 3$ and $B = 2$.
Now we iterate over each pattern in P'.
For "abr", we map "ab" to $min(2, 3 - 2) = 1$ and "br" to $min(2, 3 - 3) = 0$.
For "cad", we map "ca" to $min(2, 3 - 2) = 1$ and "ad to $min(2, 3 - 3) = 0$
For "bra", we map "br" to $min(0, 3 - 2) = 0$ and "ra" to $min(2, 3 - 3) = 0$
For "aca", we map "ac" to $min(2, 3 - 2) = 1$ and "ca" to $min(1, 3 - 3) = 0$

The table 3.2 shows the final shift table

Shift Table	
Key	Shift
ab	1
br	0
ca	0
ad	0
ra	0
ac	1

Table 3.2: *Shift table for* $\mathcal{P} = [abra, cada, bra, aca]$

3.3.3 Search

In the searching phase, we use a sliding window of size m and we calculate the polynomial rolling hash for the suffix of size B of the window and we check the Shift Table. If it's greater than zero, we shift the window accordingly and repeat the process. Otherwise we may have one, or several, matches. The hash table for this suffix key holds all possible candidates. So how do we check for them efficiently? One may do this with a brute-force fashion, checking each pattern for a match. We can do better using a heuristic described by Wu Manber and using the precomputed tables. Since the hash table value is a list of pairs where the first value is the polynomial rolling hash for the prefix, we can use it as a filter method. This heuristic is good in practice when it is unlikely to have patterns with the same prefix and suffix.

The program 3.4 shows the search phase of Wu-Manber algorithm.

Program 3.4 Wu-Manber Search

```
FUNCTION search(𝒯, 𝒫)
    idx ⟵ m−1
    while idx < |𝒯| do   ▹ Compute hash value based on current B characters from text

        hash ⟵ compute_hash(𝒯[idx − B + 1 ··· idx])

        if SHIFT[h] > 0
            idx ⟵ idx + SHIFT[h]

        else   ▹ Possible match
            prefixHash ⟵ compute_hash(𝒯[idx−m+1 ··· idx−m+B])
            for (hash, p) ∈ TABLE[h] do
                if hash = prefixHash   ▹ Wu-Manber heuristic
                    check match
            end for
            idx ⟵ idx + 1
    end while
end
```

Example

The table 3.3 shows the execution of the Wu Manber searching phase for $\mathcal{T} = abracadabra$ and $\mathcal{P} = [abra, cada, bra, aca]$, with $B = 2$. For example, at iteration 0, the shift value holds zero and the hash table is checked for the suffix br. The hash table yields the candidate abra and, since it has the same prefix value of window, we verify the match of abra, which is true.

i	window	prefix (hash)	suffix (hash)	shift
0	abr	ab (63)	br (560)	0 (match found "abra")
1	bra	br (560)	ra (49)	0 (match found "bra")
2	rac	ra (49)	ac (94)	1
3	aca	ac (94)	ca (34)	0
4	cad	ca (34)	ad (125)	0
0	abr	ab (63)	br (560)	0 (match found "abra")
1	bra	br (560)	ra (49)	0 (match found "bra")

Table 3.3: *Searching phase of Wu-Manber algorithm*

3.3.4 Complexity

Regarding the Memory complexity, the algorithm is $\mathcal{O}(|T| + |M|)$, in order to build the tables. To analyse the time-complexity, the original Wu Manber paper Wu and Manber, 1994 provides an estimation of the running time for this algorithm. The algorithm is $\mathcal{O}(B|\mathcal{T}|/|\mathcal{P}|$ of time-complexity in the average case.

Chapter 4

Comparative Analysis of Performance

4.1 Introduction

In this section, we are going to test the performance of the discussed algorithms. For testing the algorithms for the Single Pattern Matching problem, it is useful to check the performance of the algorithms against different text input sizes and pattern sizes. Therefore, we use four different texts, with increasing length (i,.e, number of words) and a set of 6 words from the text. We run the algorithms for each text and each word, and take the average measured running time for each text. For dealing with Multiple Pattern experiments, however, we use two large texts, and test the algorithms with a increasing number of patterns selected from the text. The Input texts and word lists are available in appendix 1.
All experiments were performed on a Ubuntu 20.04, Intel I5 8400, 8GBM Ram and the algorithms were implemented in C++, available at https://github.com/raphaelrbr/mac499

4.2 Single Pattern Experiments

We are going to compare the performance of the Single Pattern Matching algorithms. To do so, we conduct four experiments. In each one, we fix the test and select 8 patterns from it and we measure the running time for finding these patterns.

4.2.1 Experiment 1

For the first experiment, we use as $\mathcal{T}$ the text "Raven" by Edgar Allan Poe. We measure the running time for finding each one of the following patterns: this, that, door, chamber, bird, raven, nevermore and lenore. The results are shown in table 4.1.

Pattern	Boyer-Moore	Boyer-Moore-Horspool	Trie	Shift-Or	KMP
this	9 μs	21 μs	23997704 μs	85 μs	123 μs
that	9 μs	7 μs	24222057 μs	43 μs	132 μs
door	8 μs	8 μs	22774469 μs	42 μs	122 μs
chamber	19 μs	8 μs	22623642 μs	42 μs	112 μs
bird	9 μs	7 μs	22636409 μs	42 μs	121 μs
raven	9 μs	8 μs	22726710 μs	43 μs	127 μs
nevermore	9 μs	12 μs	22765160 μs	45 μs	119 μs
lenore	16 μs	8 μs	22682270 μs	46 μs	124 μs

Table 4.1: *Running time for the text Raven, by Edgar Allan Poe.*

4.2.2 Experiment 2

For the second experiment, we use as $\mathcal{T}$ the text "Rise" by Maya Angelou. We measure the running time for finding each one of the following patterns: rise, like, with, your, does, still, just and that. The results are shown in table 4.2.

Pattern	Boyer-Moore	Boyer-Moore-Horspool	Trie	Shift-Or	KMP
rise	3 μs	4 μs	756352 μs	22 μs	27 μs
like	3 μs	3 μs	750164 μs	18 μs	27 μs
with	7 μs	3 μs	748005 μs	37 μs	25 μs
your	7 μs	3 μs	747135 μs	18 μs	26 μs
does	5 μs	4 μs	747322 μs	18 μs	26 μs
still	6 μs	3 μs	750752 μs	20 μs	35 μs
just	8 μs	3 μs	749315 μs	19 μs	25 μs
that	7 μs	3 μs	742445 μs	20 μs	39 μs

Table 4.2: *Running time for the text Rise, by Maya Angelou.*

4.2.3 Experiment 3

For the third experiment, we use as $\mathcal{T}$ the book "DonQuixote" by Miguel de Cervantes. We measure the running time for finding each one of the following patterns: that, with, this, they, said, have, quixote and sancho. The results are shown in table 4.3. The Trie Data Structure was omitted for this experiment because the algorithm could not finish under a reasonable amount of time for the purpose of this experiment.

4.2.4 Experiment 4

For the third experiment, we use as $\mathcal{T}$ the book "Hamlet" by Shakespeare. We measure the running time for finding each one of the following patterns: hamlet, that, this, with, your, lord, what and king. The results are shown in table 4.4. Again, the Trie data structure was omitted for this experiment.

Pattern	Boyer-Moore	Boyer-Moore-Horspool	Trie	Shift-Or	KMP
that	11399 μs	9029 μs	-	11202 μs	41130 μs
with	11426 μs	7223 μs	-	11361 μs	38608 μs
this	11009 μs	7339 μs	-	11194 μs	40885 μs
they	11036 μs	7111 μs	-	11125 μs	41443 μs
said	10570 μs	7456 μs	-	11156 μs	39446 μs
have	11055 μs	7414 μs	-	111223 μs	39915 μs
quixote	8200 μs	5821 μs	-	11149 μs	37819 μs
sancho	8466 μs	5915 μs	-	11154 μs	39976 μs

Table 4.3: *Running time for the book "DonQuixote" by Miguel de Cervantes.*

Pattern	Boyer-Moore	Boyer-Moore-Horspool	Trie	Shift-Or	KMP
hamlet	942 μs	785 μs	-	859 μs	3654 μs
that	906 μs	630 μs	-	940 μs	3295 μs
this	927 μs	597 μs	-	889 μs	3349 μs
with	917 μs	587 μs	-	895 μs	3387 μs
your	943 μs	616 μs	-	914 μs	3299 μs
lord	729 μs	601 μs	-	1080 μs	3331 μs
what	923 μs	627 μs	-	897 μs	3315 μs
king	944 μs	713 μs	-	882 μs	3407 μs

Table 4.4: *Running time for the book "Hamlet" by Shakespeare.*

4.2.5 Results

As we can see, the Boyer-Moore was the fastest algorithm to find the patterns in the experiments 1 and 2. However, in the third and fourth experiment, we can see the Boyer-Moore-Horspool is slightly faster, which may suggest this algorithm can be faster than the Boyer-Moore for large texts. The Trie data structure is significantly slower than the other algorithms due to its build phase. The Shift-Or and KMP algorithms were similar in performance but still slower than the Boyer-Moore and Boyer-Moore-Horspool algorithms.

4.3 Multiple Pattern Experiments

Now, we are going to compare the performance of the Aho-Corasick algorithm and the Wu-Manber algorithm. To do so, we conduct two experiments, using a fixed text and varying the number of patterns from 8 to 256, measuring the running time to find these patterns.

4.3.1 Experiment 1

For the first experiment, we use the book "DonQuixote" by Miguel de Cervantes. We measure the running time for finding a set of patterns with length varying from 8 to 256. The patterns were chosen among the most frequent words from this book. The results are shown in table 4.5

no. of patterns	Aho-Corasick	Wu-Manber
8	168029 μs	980976 μs
16	242129 μs	1106758 μs
32	337313 μs	1321426 μs
64	891802 μs	1580664 μs
128	2329824 μs	1730832 μs
256	6037932 μs	2010092 μs

Table 4.5: *Running time for the multiple pattern algorithms for book "DonQuixote" by Miguel de Cervantes.*

4.3.2 Experiment 2

For the second experiment, we use the book "Hamlet" by Shakespeare. We measure the running time for finding a set of patterns with length varying from 8 to 256. The patterns were chosen among the most frequent words from this book. The results are shown in table 4.6

no. of patterns	Aho-Corasick	Wu-Manber
8	15841 μs	119811 μs
16	22175 μs	128568 μs
32	31451 μs	149018 μs
64	88873 μs	166448 μs
128	252300 μs	198210 μs
256	8662316 μs	214881 μs

Table 4.6: *Running time for the multiple pattern algorithms for book "Hamlet" by Shakespeare.*

4.3.3 Results

Our experiments shows that the Aho-Corasick algorithm performs faster than the Wu-Manber algorithm for a small number of patterns. However, for large number of patterns we can see the Wu-Manber algorithm performing significantly faster than the Aho-Corasick algorithm, which may suggest this algorithm is a better approach for a large number of patterns.

Chapter 5

Final Considerations

In this monograph, we have explored various algorithms for string matching and pattern recognition, including the Trie data structure, KMP, Boyer-Moore, Aho-Corasick, and Wu-Manber algorithms. These algorithms have proven to be powerful tools for solving a wide range of problems in computer science, from text processing and information retrieval to cybersecurity and data compression.

Our experiments suggest that for solving the Single Pattern Matching Problem, the Boyer-Moore and Boyer-Moore-Horspool algorithms were better suited in terms of performance. The Shift-Or and KMP were similar in performance but still slower than the first two algorithms. In last, the Trie Data Structure was the slowest due to its build phase. However, there are several ways to improve the performance of the build phase of this algorithm but these techniques are beyond the scope of this monograph.

Regarding the Multiple Pattern Matching Problem, we have studied the Wu-Manber and Aho-Corasick algorithms. Our experiments suggest that the Aho-Corasick algorithms performs faster than the Wu-Manber algorithm for a small number of patterns and the opposite for a large number of patterns. However, it is important to highlight there are many heuristics that can be used to improve the performance of these algorithms and there are many different implementations for them.

In conclusion, this monograph has provided the reader with a solid foundation in string matching algorithms and a better understanding of how these algorithms work and when to use them. These algorithms will be valuable tools for solving complex problems.

Bibliography

[Aho and Corasick 1975] Alfred V. Aho and Margaret J. Corasick. "Efficient string matching". In: *Communications of the ACM* 18.6 (1975), pp. 333–340. doi: 10.1145/360825.360855 (cit. on p. 23).

[Baeza-Yates and Gonnet 1989] R. A. Baeza-Yates and G. H. Gonnet. "A new approach to text searching". In: *ACM SIGIR Forum* 23.SI (1989), pp. 168–175. doi: 10.1145/75335.75352 (cit. on p. 15).

[Boyer and Moore 1977] Robert S. Boyer and J. Strother Moore. "A fast string searching algorithm". In: *Communications of the ACM* 20.10 (1977), pp. 762–772. doi: 10.1145/359842.359859 (cit. on p. 4).

[Horspool 1980] R. Nigel Horspool. "Practical fast searching in strings". In: *Software: Practice and Experience* 10.6 (1980), pp. 501–506. doi: 10.1002/spe.4380100608 (cit. on p. 8).

[Knuth *et al.* 1977] Donald Ervin Knuth, Vaughan R. Pratt, and James H. Morris. *Fast pattern matching in strings*. s.n., 1977 (cit. on p. 12).

[Wu and Manber 1994] S. Wu and U. Manber. *A fast algorithm for multi-pattern searching*. University of Arizona. Department of Computer Science, 1994 (cit. on pp. 28, 32).